Yale University
Department of Computer Science

Building Certified Libraries for PCC:
Dynamic Storage Allocation

Dachuan Yu Nadeem A. Hamid Zhong Shao
Yale University

YALEU/DCS/TR-1247
January 13, 2003

This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

Report Documentation Page	*Form Approved OMB No. 0704-0188*

1. REPORT DATE **13 JAN 2003**	2. REPORT TYPE	3. DATES COVERED **00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE **Building Certified Libraries for PCC: Dynamic Storage Allocation**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Yale University,Department of Computer Science,New Haven,CT,06520**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Proof-Carrying Code (PCC) allows a code producer to provide to a host a program along with its formal safety proof. The proof attests a certain safety policy enforced by the code, and can be mechanically checked by the host. While this language-based approach to code certification is very general in principle existing PCC systems have only focused on programs whose safety proofs can be automatically generated. As a result, many low-level system libraries (e.g. memory management) have not yet been handled. In this paper, we explore a complementary approach in which general properties and program correctness are semi-automatically certified. In particular, we introduce a low-level language CAP for building certified programs and present a certified library for dynamic storage allocation.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**22**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

Proof-Carrying Code (PCC) allows a code producer to provide to a host a program along with its formal safety proof. The proof attests a certain safety policy enforced by the code, and can be mechanically checked by the host. While this language-based approach to code certification is very general in principle, existing PCC systems have only focused on programs whose safety proofs can be automatically generated. As a result, many low-level system libraries (*e.g.,* memory management) have not yet been handled. In this paper, we explore a complementary approach in which general properties and program correctness are semi-automatically certified. In particular, we introduce a low-level language CAP for building certified programs and present a certified library for dynamic storage allocation.

Building Certified Libraries for PCC: Dynamic Storage Allocation

Dachuan Yu Nadeem A. Hamid Zhong Shao

Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{yu,hamid-nadeem,shao}@cs.yale.edu
Technical Report YALEU/DCS/TR-1247
January 13, 2003

Abstract. Proof-Carrying Code (PCC) allows a code producer to provide to a host a program along with its formal safety proof. The proof attests a certain safety policy enforced by the code, and can be mechanically checked by the host. While this language-based approach to code certification is very general in principle, existing PCC systems have only focused on programs whose safety proofs can be automatically generated. As a result, many low-level system libraries (*e.g.,* memory management) have not yet been handled. In this paper, we explore a complementary approach in which general properties and program correctness are semi-automatically certified. In particular, we introduce a low-level language CAP for building certified programs and present a certified library for dynamic storage allocation.

1 Introduction

Proof-Carrying Code (PCC) is a general framework pioneered by Necula and Lee [15, 13]. It allows a code producer to provide a program to a host along with a formal safety proof. The proof is incontrovertible evidence of safety which can be mechanically checked by the host; thus the host can safely execute the program even though the producer may not be trusted.

Although the PCC framework is general and potentially applicable to certifying arbitrary data objects with complex specifications [14, 2], generating proofs remains difficult. Existing PCC systems [16, 12, 3, 1] have only focused on programs whose safety proofs can be automatically generated. As a result, many low-level system libraries, such as dynamic storage allocation, have not been certified. Nonetheless, building certified libraries, especially low-level system libraries, is an important task in certifying compilation. It not only helps increase

This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

the reliability of "infrastructure" software by reusing provably correct program routines, but also is crucial in making PCC scale for production.

On the other hand, Hoare logic [7, 8], as a widely applied approach in program verification, allows programmers to express their reasonings with assertions and the application of inference rules, and can be used to prove general program correctness. In this paper, we introduce a conceptually simple low-level language for certified assembly programming (CAP) that supports Hoare-logic style reasoning. We use CAP to build a certified library for dynamic storage allocation, and further use this library to build a certified program whose correctness proof can be mechanically checked. Applying Hoare-logic reasonings at an assembly-level, our paper makes the following contributions:

- CAP is based on a common instruction set so that programs can be executed on real machines with little effort. The expected behavior of a program is explicitly written as a specification using higher-order logic. The programmer proves the well-formedness of a program with respect to its specification using logic reasoning, and the result can be checked mechanically by a proof-checker. The soundness of the language guarantees that if a program passes the static proof-checking, its run-time behavior will satisfy the specification.
- Using CAP, we demonstrate how to build certified libraries and programs. The specification of library routines are precise yet general enough to be imported in various user programs. Proving the correctness of a user program involves linking with the library proofs.
- We implemented CAP and the dynamic storage allocation routines using the Coq proof assistant [20], showing that this library is indeed certified. The example program is also implemented. All the Coq code is available [21].
- Lastly, memory management is an important and error-prone part of most non-trivial programs. It is also considered to be hard to certify by previous researches. We present a provably correct implementation of a typical dynamic storage allocation algorithm. To the authors' knowledge, it is so far the only certified library for memory management.

2 Dynamic Storage Allocation

In the remainder of this paper, we focus on the certification and use of a library module for dynamic storage allocation. In particular, we implement a storage allocator similar to that described in [10, 11]. The interface to our allocator consists of the standard `malloc` and `free` functions. The implementation keeps track of a *free list* of blocks which are available to satisfy memory allocation requests. As shown in Figure 1, the free list is a null-terminated list of (non-contiguous) memory blocks. Each block in the list contains a header of two words: the first stores a pointer to the next block in the list, and the second stores the size of the block. The allocated block pointer that is returned to a user program points to the useable space in the block, not to the header.

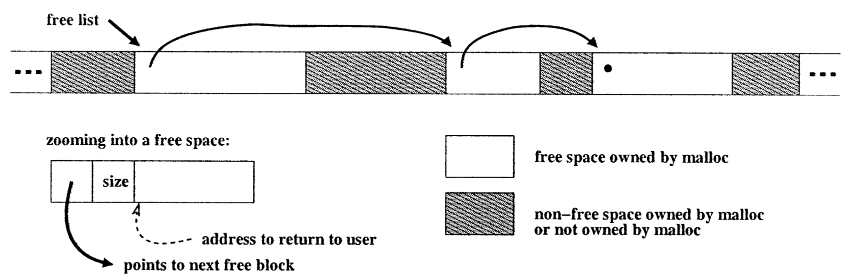

Fig. 1. Free list and free blocks.

The blocks in the list are sorted in order of increasing address and requests for allocation are served based on a first-fit policy; hence, we implement an *address-ordered first-fit* allocation mechanism. If no block in the free list is big enough, or if the free list is empty, then `malloc` requests more memory from the operating system as needed. When a user program is done with a memory block, it is returned to the free list by calling `free`, which puts the memory block into the free list at the appropriate position.

Our implementation in this paper is simple enough to understand, yet faithfully represents mechanisms used in traditional implementations of memory allocators [22, 10, 11]. For ease of presentation, we assume our machine never runs out of memory so `malloc` will never fail, but otherwise many common low-level mechanisms and techniques used in practice are captured in this example, such as use of a free list, in-place header fields, searching and sorting, and splitting and coalescing (described below). We thus believe our techniques can be as easily applied to a variety of other allocator implementations than described here.

In the remainder of this section, we describe in detail the functionality of the `malloc` and `free` library routines (Figure 2), and give some "pseudo-code" for them. We do not show the `calloc` (allocate and initialize) and `realloc` (resize allocated block) routines because they essentially delegate their tasks to the two main functions described below.

`free` This routine puts a memory block into the free list. It takes a pointer (`ptr`) to the useable portion of a memory block (preceded by a valid header) and does not return anything. It relies on the preconditions that `ptr` points to a valid "memory block" and that the free list is currently in a good state (*i.e.*, properly sorted). As shown in Figure 2, `free` works by walking down the free list to find the appropriate (address-ordered) position for the block. If the block being freed is directly adjacent with either neighbor in the free list, the two are *coalesced* to form a bigger block.

`malloc` This routine is the actual storage allocator. It takes the size of the new memory block expected by the user program, and returns a pointer to an available block of memory of that size. As shown in Figure 2, `malloc` calculates

```
void free (void* ptr) {
   hp = ptr - header_size;                                    // move to header
   for (prev = nil, p = flist; p <> nil; prev = p, p = p->next)
      if (hp < p) {                                               // found place
         if (hp + hp->size == p)      // join or link with upper neighbor
              hp->size += p->size, hp->next = p->next;
         else hp->next = p;
         if (prev <> nil)             // join or link with lower neighbor
              if (prev + prev->size == hp)
                   prev->size += hp->size, prev->next = hp->next;
              else prev->next = hp;
         else flist = hp;
         return;
      }
   hp->next = nil;                    // block's place is at end of the list
   if (prev <> nil)                      // join or link with lower neighbor
      if (prev + prev->size == hp)
         prev->size += hp->size, prev->next = hp->next;
      else prev->next = hp;
   else flist = hp;
}
void* malloc (int reqsize) {
   actual_size = reqsize + header_size;
   for(prev = nil, p = flist; ; prev = p, p = p->next)
      if (p==nil) {                 // end of free list, request more memory
         more_mem(actual_size);
         prev = nil, p = flist;                   // restart the loop search
      } else if (p->size > actual_size + header_size) {
         p->size -= actual_size;         // found block bigger than needed
         p += p->size;                   //    by more than a header size,
         p->size = actual_size;          //                 so split into two
         return (p + header_size);
      } else if (p->size >= actual_size) {      // found good enough block
         if (prev==nil) flist = p->next; else prev->next = p->next;
         return (p + header_size);
      }
}
void more_mem(int req_size) {
  if (req_size < NALLOC) req_size = NALLOC;       // request not too small
  q = alloc(req_size);                            // call system allocator
  q->size = req_size;
  free(q + header_size);                       // put new block on free list
}
```

Fig. 2. Pseudo code of allocation routines.

the actual size of the block needed including the header and then searches the free list for the first available block with size greater than or equal to what is required. If the size of the block found is large enough, it is *split* into two and a pointer to the tail end is returned to the user.

If no block in the free list is large enough to fulfill the request, more memory is requested from the system by calling `more_mem`. Because this is a comparatively expensive operation, `more_mem` requests a minimum amount of memory each time to reduce the frequency of these requests. After getting a new chunk of memory from the system, it is appended onto the free list by calling `free`.

These dynamic storage allocation algorithms often temporarily break certain invariants, which makes it hard to automatically prove their correctness. During intermediate steps of splitting, coalescing, or inserting memory blocks into the free list, the state of the free list or the memory block is not valid for one or two instructions. Thus, a traditional type system would need to be extremely specialized to be able to handle such code.

3 A Language for Certified Assembly Programming (CAP)

To write our certified libraries, we use a low-level assembly language CAP fitted with specifications reminiscent of Hoare-logic. The assertions that we use for verifying the particular dynamic allocation library described in this paper are inspired by Reynolds' "separation logic" [19, 18].

The syntax of CAP is given in Figure 3. A complete program (or, more accurately, machine state) consists of a code heap, a dynamic state component made up of the register file and data heap, and an instruction sequence. The instruction set captures the most basic and common instructions of an assembly language, and includes primitive alloc and free commands which are to be viewed as system calls. The register file is made up of 32 registers and we assume an unbounded heap with integer words of unlimited size for ease of presentation.

Our type system, as it were, is a very general layer of specifications such that assertions can be associated with programs and instruction sequences. Our assertion language (*Assert*) is the calculus of inductive constructions (CiC) [20, 17], an extension of the calculus of constructions [4] which is a higher-order typed lambda calculus that corresponds to higher-order predicate logic via the Curry-Howard isomorphism [9]. In particular, we implement the system described in this paper using the Coq proof assistant [20]. Assertions are thus defined as Coq terms of type $\mathsf{State} \to \mathsf{Prop}$, where the various syntactic categories of the assembly language (such as State) have been encoded using inductive definitions. We give examples of inductively defined assertions used for reasoning about memory in later sections.

3.1 Operational Semantics

The operational semantics of the assembly language is fairly straightforward and is defined in Figures 4 and 5. The former figure defines a "macro" relation detailing the effect of simple instructions on the dynamic state of the machine.

$$
\begin{array}{rrl}
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{C}, \mathbb{S}, \mathbb{I}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \leadsto \mathbb{I}\}^* \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{H}, \mathbb{R}) \\
(\mathit{Heap}) & \mathbb{H} & ::= \{\mathtt{l} \leadsto \mathtt{w}\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \leadsto \mathtt{w}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= \{\mathtt{r}_k\}^{k \in \{0 \ldots 31\}} \\
(\mathit{Labels}) & \mathtt{f}, \mathtt{l} & ::= i \ (\mathit{nat\ nums}) \\
(\mathit{WordVal}) & \mathtt{w} & ::= i \ (\mathit{nat\ nums}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \mathsf{c}; \mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{r}
\end{array}
\qquad
\begin{array}{rrl}
(\mathit{Command}) & \mathsf{c} & ::= \mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i \\
& & \mid \mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{subi}\ \mathtt{r}_d, \mathtt{r}_s, i \\
& & \mid \mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s \mid \mathsf{movi}\ \mathtt{r}_d, i \\
& & \mid \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \mid \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f} \\
& & \mid \mathsf{alloc}\ \mathtt{r}_d[\mathtt{r}_s] \mid \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(i) \\
& & \mid \mathsf{st}\ \mathtt{r}_d(i), \mathtt{r}_s \mid \mathsf{free}\ \mathtt{r}_s \\
(\mathit{CdHpSpec}) & \Psi & ::= \{\mathtt{f} \leadsto \mathsf{a}\}^* \\
(\mathit{Assert}) & \mathsf{a} & ::= \ldots
\end{array}
$$

Fig. 3. Syntax of CAP.

Control-flow instructions, such as jd or bgt, do not affect the data heap or register file. The domain of the heap is altered by either an alloc command, which increases the domain with a specified number of labels mapped to undefined data, or by free, which removes a label from the domain of the heap. The ld and st commands are used to access or update the value stored at a given label.

Since we intend to model realistic low-level assembly code, we do not have a "halt" instruction. In fact, termination is undesirable since it means the machine has reached a "stuck" state where, for example, a program is trying to branch to a non-existent code label, or access an invalid data label. We present in the next section a system of inference rules for specifications which allow one to statically prove that a program will never reach such a bad state.

3.2 Inference Rules

We define a set of inference rules allowing us to prove specification judgments of the following forms:

$\Psi \vdash \{\mathsf{a}\}\, \mathbb{P}$ (*well-formed program*)
$\Psi \vdash \mathbb{C}$ (*well-formed code heap*)
$\Psi \vdash \{\mathsf{a}\}\, \mathbb{I}$ (*well-formed instruction sequence*)

Programs in our assembly language are written in continuation-passing style because there are no call/return instructions. Hence, we only specify preconditions for instruction sequences. (Preconditions of the continuations actually serve as the postconditions.) If a given state satisfies the precondition, the sequence of instructions will run without reaching a bad state. Furthermore, in order to check code blocks, which are potentially mutually recursive, we require that all labels in the code heap be associated with a precondition– this mapping is our code heap specification, Ψ.

Well-formed code heap and programs A code heap is well-formed if the code block associated with every label in the heap is well-formed under the corresponding precondition. Then, a complete program is well-formed if the code heap is well-formed, the current instruction sequence is well-formed under the precondition,

if $\mathsf{c} =$	then $\mathtt{AuxStep}(\mathsf{c}, (\mathbb{H}, \mathbb{R})) =$
$\mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$
$\mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) + i\})$
$\mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$
$\mathsf{subi}\ \mathtt{r}_d, \mathtt{r}_s, i$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) - i\})$
$\mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)\})$
$\mathsf{movi}\ \mathtt{r}_d, \mathtt{w}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathtt{w}\})$

Fig. 4. Auxiliary state update macro.

and the precondition also holds for the dynamic state.

$$\frac{\mathrm{dom}(\Psi) = \mathrm{dom}(\mathbb{C}) \qquad \Psi \vdash \{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f}) \quad \forall f \in \mathrm{dom}(\Psi)}{\Psi \vdash \mathbb{C}} \tag{1}$$

$$\frac{\Psi \vdash \mathbb{C} \qquad \Psi \vdash \{\mathsf{a}\}\,\mathbb{I} \qquad (\mathsf{a}\ \mathbb{S})}{\Psi \vdash \{\mathsf{a}\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})} \tag{2}$$

Well-formed instructions: Pure rules The inference rules for instruction sequences can be divided into two categories: pure rules, which do not interact with the data heap, and impure rules, which deal with access and modification of the data heap.

The structure of many of the pure rules is very similar. They involve showing that for all states, if an assertion a holds, then there exists an assertion a′ which holds on the state resulting from executing the current command and, additionally, the remainder of the instruction sequence is well-formed under a′. We use the auxiliary state update macro defined in Figure 4 to collapse the rules for arithmetic instructions into a single schema. For control flow instructions, we instead require that if the current assertion a holds, then the precondition of the label that is being jumped to must also be satisfied.

$$\frac{\begin{array}{l}\mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{sub}, \mathsf{subi}, \mathsf{mov}, \mathsf{movi}\} \\ \forall \mathbb{H}.\ \forall \mathbb{R}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}'\ (\mathtt{AuxStep}(\mathsf{c}, (\mathbb{H}, \mathbb{R}))) \qquad \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I}\end{array}}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{c}; \mathbb{I}} \tag{3}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\ \forall \mathbb{R}.\ (\mathbb{R}(\mathtt{r}_s) \leq \mathbb{R}(\mathtt{r}_t)) \supset \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}'\ (\mathbb{H}, \mathbb{R}) \\ \forall \mathbb{H}.\ \forall \mathbb{R}.\ (\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)) \supset \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}_1\ (\mathbb{H}, \mathbb{R}) \\ \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I} \qquad \Psi(\mathtt{f}) = \mathsf{a}_1\end{array}}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}} \tag{4}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\ \forall \mathbb{R}.\ (\mathbb{R}(\mathtt{r}_s) \leq i) \supset \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}'\ (\mathbb{H}, \mathbb{R}) \\ \forall \mathbb{H}.\ \forall \mathbb{R}.\ (\mathbb{R}(\mathtt{r}_s) > i) \supset \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}_1\ (\mathbb{H}, \mathbb{R}) \\ \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I} \qquad \Psi(\mathtt{f}) = \mathsf{a}_1\end{array}}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}} \tag{5}$$

$$\frac{\forall \mathbb{S}.\ \mathsf{a}\ \mathbb{S} \supset \mathsf{a}_1\ \mathbb{S} \quad \text{where } \Psi(\mathtt{f}) = \mathsf{a}_1}{\Psi \vdash \{\mathsf{a}\}\,\mathsf{jd}\ \mathtt{f}} \tag{6}$$

	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \longmapsto \mathbb{P}$ where
if $\mathbb{I} =$	then $\mathbb{P} =$
$\mathsf{jd}\ \mathtt{f}$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}'$
$\mathsf{jmp}\ \mathtt{r}$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ where $\mathbb{C}(\mathbb{R}(\mathtt{r})) = \mathbb{I}'$
$\mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \leq \mathbb{R}(\mathtt{r}_t)$; and $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}'')$ when $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}''$
$\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \leq i$; and $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}'')$ when $\mathbb{R}(\mathtt{r}_s) > i$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}''$
$\mathsf{alloc}\ \mathtt{r}_d[\mathtt{r}_s]; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}', \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\}), \mathbb{I}')$ where $\mathbb{R}(\mathtt{r}_s) = i$, $\mathbb{H}' = \mathbb{H}\{\mathtt{l} \rightsquigarrow \_, \ \ldots, \ \mathtt{l} + i - 1 \rightsquigarrow \_\}$ and $\{\mathtt{l}, \ldots, \mathtt{l} + i - 1\} \cap \mathrm{dom}(\mathbb{H}) = \emptyset$
$\mathsf{free}\ \mathtt{r}_s; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}', \mathbb{R}), \mathbb{I}')$ where $\forall \mathtt{l} \in \mathrm{dom}(\mathbb{H}').\mathbb{H}'(\mathtt{l}) = \mathbb{H}(\mathtt{l})$, $\mathbb{R}(\mathtt{r}_s) \in \mathrm{dom}(\mathbb{H})$, and $\mathrm{dom}(\mathbb{H}') = \mathrm{dom}(\mathbb{H}) - \mathbb{R}(\mathtt{r}_s)$
$\mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(i); \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + i)\}), \mathbb{I}')$ where $(\mathbb{R}(\mathtt{r}_s) + i) \in \mathrm{dom}(\mathbb{H})$
$\mathsf{st}\ \mathtt{r}_d(i), \mathtt{r}_s; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}\{\mathbb{R}(\mathtt{r}_d) + i \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{R}), \mathbb{I}')$ where $(\mathbb{R}(\mathtt{r}_d) + i) \in \mathrm{dom}(\mathbb{H})$
$\mathsf{c}; \mathbb{I}'$ for remaining cases of c	$(\mathbb{C}, \mathtt{AuxStep}(\mathsf{c}, (\mathbb{H}, \mathbb{R})), \mathbb{I}')$

Fig. 5. Operational semantics.

$$\frac{\forall \mathbb{H}.\ \forall \mathbb{R}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}_1\ (\mathbb{H}, \mathbb{R}) \quad \text{where } \Psi(\mathbb{R}(\mathtt{r})) = \mathsf{a}_1}{\Psi \vdash \{\mathsf{a}\}\, \mathsf{jmp}\ \mathtt{r}} \tag{7}$$

Well-formed instructions: Impure rules As mentioned previously, these rules involve accessing or modifying the data heap.

$$\frac{\begin{array}{l}\forall \mathbb{H}.\ \forall \mathbb{R}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset \mathsf{a}'\ (\mathbb{H}\{\mathtt{l} \rightsquigarrow \_, \ \ldots, \ \mathtt{l} + i - 1 \rightsquigarrow \_\}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\}) \\ \text{where } \mathbb{R}(\mathtt{r}_s) = i \text{ and } \{\mathtt{l}, \ldots, \mathtt{l} + i - 1\} \cap \mathrm{dom}(\mathbb{H}) = \emptyset \\ \Psi \vdash \{\mathsf{a}'\}\, \mathbb{I}\end{array}}{\Psi \vdash \{\mathsf{a}\}\, \mathsf{alloc}\ \mathtt{r}_d[\mathtt{r}_s]; \mathbb{I}} \tag{8}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\ \forall \mathbb{R}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset ((\mathbb{R}(\mathtt{r}_s) + i) \in \mathrm{dom}(\mathbb{H})) \wedge (\mathsf{a}'\ (\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + i)\})) \\ \Psi \vdash \{\mathsf{a}'\}\, \mathbb{I}\end{array}}{\Psi \vdash \{\mathsf{a}\}\, \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(i); \mathbb{I}} \tag{9}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\ \forall \mathbb{R}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset ((\mathbb{R}(\mathtt{r}_d) + i) \in \mathrm{dom}(\mathbb{H})) \wedge (\mathsf{a}'\ (\mathbb{H}\{\mathbb{R}(\mathtt{r}_d) + i \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{R})) \\ \Psi \vdash \{\mathsf{a}'\}\, \mathbb{I}\end{array}}{\Psi \vdash \{\mathsf{a}\}\, \mathsf{st}\ \mathtt{r}_d(i), \mathtt{r}_s; \mathbb{I}} \tag{10}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\ \forall \mathbb{R}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}) \supset (\mathbb{R}(\mathtt{r}_s) \in \mathrm{dom}(\mathbb{H})) \wedge (\mathsf{a}'\ (\mathbb{H}', \mathbb{R})) \\ \text{where } \mathrm{dom}(\mathbb{H}') = \mathrm{dom}(\mathbb{H}) - \mathbb{R}(\mathtt{r}_s) \text{ and } \forall \mathtt{l} \in \mathrm{dom}(\mathbb{H}').\mathbb{H}'(\mathtt{l}) = \mathbb{H}(\mathtt{l}) \\ \Psi \vdash \{\mathsf{a}'\}\, \mathbb{I}\end{array}}{\Psi \vdash \{\mathsf{a}\}\, \mathsf{free}\ \mathtt{r}_s; \mathbb{I}} \tag{11}$$

3.3 Soundness

We establish the soundness of these inference rules with respect to the operational semantics of the machine following the syntactic approach of proving type soundness [23]. From "Type Preservation" and "Progress" lemmas (proved by induction on $\mathbb{I}$), we can guarantee that given a well-formed program, the current instruction sequence will be able to execute without getting "stuck." Furthermore, at the point when the current instruction sequence branches to another code block, the machine state will always satisfy the precondition of that block.

Lemma 1 (Type Preservation). *If $\Psi \vdash \{a\}\,(\mathbb{C},\mathbb{S},\mathbb{I})$ and $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto \mathbb{P}$, then there exists an assertion a' such that $\Psi \vdash \{a'\}\,\mathbb{P}$.*

Lemma 2 (Progress). *If $\Psi \vdash \{a\}\,(\mathbb{C},\mathbb{S},\mathbb{I})$, then there exists a program $\mathbb{P}$ such that $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto \mathbb{P}$.*

Theorem 1 (Soundness). *If $\Psi \vdash \{a\}\,(\mathbb{C},\mathbb{S},\mathbb{I})$, then there exists a program $\mathbb{P}$ such that $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto \mathbb{P}$, and*

- *if $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^* (\mathbb{C},\mathbb{S}',\mathsf{jd}\ f)$, then $\Psi(f)\ \mathbb{S}'$;*
- *if $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^* (\mathbb{C},(\mathbb{H},\mathbb{R}),\mathsf{jmp}\ r_d)$, then $\Psi(\mathbb{R}(r_d))\ (\mathbb{H},\mathbb{R})$;*
- *if $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^* (\mathbb{C},(\mathbb{H},\mathbb{R}),(\mathsf{bgt}\ r_s,r_t,f))$ and $\mathbb{R}(r_s) > \mathbb{R}(r_t)$, then $\Psi(f)\ (\mathbb{H},\mathbb{R})$;*
- *if $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^* (\mathbb{C},(\mathbb{H},\mathbb{R}),(\mathsf{bgti}\ r_s,i,f))$ and $\mathbb{R}(r_s) > i$, then $\Psi(f)\ (\mathbb{H},\mathbb{R})$.*

It should be noted here that this soundness theorem establishes more than simple type safety. In addition to that, it states that whenever we jump to a block of code in the heap, the specified precondition of that code (which is an arbitrary assertion) will hold.

4 Certified Dynamic Storage Allocation

Equipped with CAP, we are ready to build the certified library. In particular, we provide provably correct implementation for the library routines `free` and `malloc`. The main difficulties involved in this task are: (1) to give precise yet general specifications to the routines; (2) to prove as theorems the correctness of the routines with respect to their specifications; (3) the specifications and theorems have to be modular so that they can interface with user programs. In this section, we discuss these problems for `free` and `malloc` respectively. From now on, we use the word "specification" in the wider sense, meaning anything that describes the behavior of a program. To avoid confusion, we call the language construct Ψ a *code heap spec*, or simply *spec*.

Before diving into certifying the library, we define some assertions related to memory blocks and the free list as shown in Figure 6. These definitions make use of some basic operators (which we implement as shorthands using primitive constructs) commonly seen in separation logic [19, 18]. In particular, **emp** asserts that the heap is empty; $e \mapsto e'$ asserts that the heap contains one cell at address e which contains e'; and separating conjunction $p{*}q$ asserts that the heap can be split into two disjoint parts in which p and q hold respectively.

Memory block ($\mathsf{MBlk}\ p\ q\ s$) asserts that the memory at address p is preceded by a pair of words: the first word contains q, a (possibly null) pointer to another

$$
\begin{aligned}
&\mathsf{MBlk}\ p\ q\ s\\
&\quad \equiv (p > 2) \wedge (s > 2) \wedge\\
&\qquad (p-2 \mapsto q) * (p-1 \mapsto s)\\
&\qquad * (p, \ldots, p+s-3 \mapsto \_, \ldots, \_)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{MBlkLst}\ 0\ p\ q\\
&\quad \equiv \mathbf{emp} \wedge (p = q)\\
&\mathsf{MBlkLst}\ (n+1)\ p\ q\\
&\quad \equiv \exists p'.(\mathsf{MBlk}\ (p+2)\ p'\ \_)\\
&\qquad\quad * (\mathsf{MBlkLst}\ n\ p'\ q)\\
&\qquad\quad \wedge (p < p' \vee p' = \mathtt{nil})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{EndL}\ \mathit{flist}\ p\ q\\
&\quad \equiv ((p = \mathtt{nil}) \supset (\mathsf{MBlkLst}\ 0\ \mathit{flist}\ q))\\
&\qquad \wedge (p \neq \mathtt{nil}\\
&\qquad\quad \supset \exists n.\ ((\mathsf{MBlkLst}\ n\ \mathit{flist}\ p)\\
&\qquad\qquad\quad * (\mathsf{MBlk}\ (p+2)\ q\ \_)\\
&\qquad\qquad\quad \wedge (p < q \vee q = \mathtt{nil})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{MidL}\ \mathit{flist}\ p\ q\\
&\quad \equiv \exists n.(\mathsf{EndL}\ \mathit{flist}\ p\ q)\\
&\qquad\quad * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Good}\ \mathit{flist}\\
&\quad \equiv \exists n.(\mathsf{MBlkLst}\ n\ \mathit{flist}\ \mathtt{nil})
\end{aligned}
$$

$$
\forall p.\forall q.\ (\mathsf{MidL}\ \mathit{flist}\ p\ q) \supset (\mathsf{Good}\ \mathit{flist})
$$

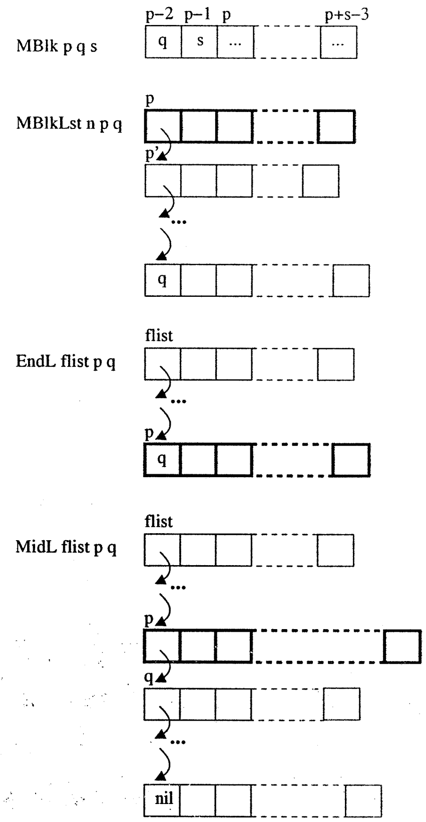

Fig. 6. Assertions on free list.

memory block, and the second word contains the size of the memory block itself (including the two-word header preceding p).

Memory block list ($\mathsf{MBlkLst}\ n\ p\ q$) models an address-ordered list of blocks. n is the number of blocks in the list, p is the starting pointer and q is the ending pointer. This assertion is defined inductively and is a specialized version of the *singly-linked list* introduced by Reynolds [19, 18]. However, unlike the somewhat informal definition of singly-linked list, MBlkLst has to be defined formally for mechanical proof-checking. Thus we use a Coq inductive definition for this purpose. In contrast, if the assertion language is defined syntactically, inductive definitions have to be defined in the assertion language, which is not shown in previous work.

A list with ending block ($\mathsf{EndL}\ \mathit{flist}\ p\ q$) is defined as a list flist of memory blocks with p pointing at the last block whose forward pointer is q. In the special case that flist is an empty list, p and q are $\mathtt{nil}$. ($\mathsf{MidL}\ \mathit{flist}\ p\ q$) models a list with a block B in the middle, where the list starts from flist, and the block B is specified by the position p and the forward pointer q. This assertion is defined as the separating conjunction of a list with ending block B and a null-terminated list starting from the forward pointer of B.

Finally we define a good free list (Good) as a null-terminated memory block list. It is easy to show the relation between MidL and Good as described.

`free` Putting aside the syntax for the moment, a specification which models the expected behavior of `free` can be written as the following Hoare triple:

$$\begin{array}{l}\{PRE\}\ \texttt{free}(fptr)\ \{POST\};\\ \text{where } PRE \equiv Pred * (\mathsf{MBlk}\ fptr\ \_\ \_) * (\mathsf{Good}\ flist)\\ \qquad POST \equiv Pred * (\mathsf{Good}\ flist)\end{array}$$

Assertion PRE states the precondition. It requires that the heap can be separated into three disjoint parts, where $fptr$ points to a memory block to be freed; $flist$ points to a good free list; and the remaining part satisfies the user specified assertion $Pred$. Assertion $POST$ states the postcondition. Since the memory block is placed into the free list, the heap now can be separated into two disjoint parts: $flist$ still points to a good free list, and the remaining part of the heap still satisfies $Pred$ because it is untouched.

Note that this does not totally specify all the behaviors of `free`. For example, it is possible to add in the postcondition that the memory block that $fptr$ pointed to is now in the free list. However, this is irrelevant from a library user's point of view. Thus we favor the above specification, which guarantees that `free` does not affect the remaining part of the heap.

Now we write this specification in CAP, where programs are written in continuation-passing style. Before `free` completes its job and jumps to the return pointer, the postcondition should be established. Thus the postcondition can be interpreted as the precondition of the code referred to by the return pointer. Suppose $\mathtt{r}_0$ is the return pointer, a valid library call to `free` should require that $POST$ implies $\Psi(\mathbb{R}(\mathtt{r}_0))$ for all states (which we write as $POST \Longrightarrow \Psi(\mathbb{R}(\mathtt{r}_0))$). In fact, this condition is required for type-checking the returning code of `free` (*i.e.,* jmp $\mathtt{r}_0$). As a library routine, `free` is expected to be used in various programs with different code heap specs (Ψ). So the above condition has to be established by the user with the actually knowledge of Ψ. When proving the well-formedness of `free`, this condition is taken as a premise.

At an assembly-level, most non-trivial programs are expressed as multiple code blocks connected together with control flow instructions (jd, jmp and bgt). Type-checking these control flow instructions requires similar knowledge about the code heap spec Ψ. For instance, at the end of the code block `free`, an assertion A_{iter} is established about the current state, and the control is transferred to the code block `iter` with a direct jump. When type-checking this direct jump (*i.e.,* jd `iter`) against the assertion A_{iter}, the inference rule 6 requires that A_{iter} implies $\Psi(\mathtt{iter})$ for all states. These requirements are also taken as premises in the well-formedness theorem of `free`. Thus the specification of `free` is actually as follows:

$$\begin{array}{l}\forall Pred.\, \forall\Psi.\, \forall \mathtt{f}.\, (POST \Longrightarrow \Psi(\mathtt{f})) \wedge (A_{iter} \Longrightarrow \Psi(\mathtt{iter}))\\ \qquad\qquad \supset \Psi \vdash \{PRE \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}\, \mathbb{C}(\mathtt{free})\end{array}$$

where $\mathbb{C}(\texttt{free})$ is the code block labeled `free`, $\mathtt{r}_0$ holds the return location, and universally quantified *Pred* occurs inside the macros *PRE* and *POST* as defined before. This is defined as a theorem and formally proved in Coq.

Following similar ideas, the well-formedness of all the other code blocks implementing the library routine `free` are also modeled and proved as theorems, with the premises changed appropriately according to which labels they refer to.

Using the Coq proof-assistant, proving these theorems is not difficult. Pure instructions only affect the register file; they are relatively easy to handle. Impure instructions affect the heap. Nonetheless, commonalities on similar operations can be factored out as lemmas. For instance, writing into the "link" field of a memory block header occurs in various places. By factoring out this behavior as a lemma and applying it, the proof construction becomes simple routine work. The only tricky part lies in proving the code which performs coalescing of free blocks. This operation essentially consists of two steps: one to modify the size field; the other to combine the blocks. No matter which one is performed first, one of the blocks has to be "broken" from being a valid memory block as required by MBlk. This behavior is hard to handle in conventional type systems, because it tends to break certain invariants captured by the type system.

In Figure 9 of Appendix A, we give the routine `free` written in CAP. This program is annotated with assertions at various program points. It contains the spec templates (the assertions at the beginning of every code block), and can be viewed as an outline of the proof. In this program, variables are used instead of register names for ease of understanding. We also assume all registers to be caller-saved, so that updating the register file does not affect the user customized assertion *Pred*. Typically relevant states are saved in activation records in a stack when making function calls, and *Pred* would be dependent only on the stack. In the current implementation, we have not yet provided certified activation records; instead, we simply use different registers for different programs.

A certified library routine consists of both the code and the proof. Accordingly, the interface of such a routine consists of both the signature (parameters) and the spec templates (*e.g., PRE,POST*). When the routine is used by a user program, both the parameters and the spec templates should be instantiated properly. The well-formedness of `free` is also a template which can be applied to various assertion *Pred*, code heap spec Ψ and returning label f. If a user program contains only one call-site to `free`, the corresponding assertion for `free` should be used in Ψ. However, if a user program contains multiple call-sites to `free`, a "sufficiently weak" assertion for `free` must be constructed by building a disjunction of all the individually instantiated assertions. The following derived Rule 12 (which is proved by induction on $\mathbb{I}$), together with the theorem for the well-formedness of `free`, guarantees that the program type-checks.

$$\frac{\Psi \vdash \{\mathrm{a}_1\}\,\mathbb{I} \qquad \Psi \vdash \{\mathrm{a}_2\}\,\mathbb{I}}{\Psi \vdash \{\mathrm{a}_1 \vee \mathrm{a}_2\}\,\mathbb{I}} \tag{12}$$

`malloc` Similarly as for `free`, an informal specification of `malloc` can be described as follows:

$$\{PRE\}\ \mathtt{malloc}(nsize, mptr)\ \{POST\};$$
$$\text{where } PRE \equiv Pred * (\mathsf{Good}\ flist) \wedge (nsize = s_0 > 0)$$
$$POST \equiv Pred' * (\mathsf{Good}\ flist) * (\mathsf{MBlk}\ mptr\ \_\ s) \wedge\ (s_0 + 2 \leq s)$$

The precondition PRE states that $flist$ points to a good free list, user customized assertion $Pred$ holds for the remaining part of the heap, and the requested size $nsize$ is larger than 0. The postcondition $POST$ states that part of the heap is the newly allocated memory block pointed to by $mptr$ whose size is at least the requested one, $flist$ still points to a good free list, and another assertion $Pred'$ holds for the remaining part of the heap. $Pred'$ may be different from $Pred$ because `malloc` modifies register $mptr$. The relation between these two assertions is described by $SIDE$ as follows:

$$SIDE \equiv \forall(\mathbb{H}, \mathbb{R}).\ Pred\ (\mathbb{H}, \mathbb{R}) \supset Pred'\ (\mathbb{H}, \mathbb{R}\{mptr \rightsquigarrow \_\})$$

Typically, $Pred$ does not depend on $mptr$. So $Pred'$ is the same as $Pred$ and the above condition is trivially established.

To type-check the control-flow instructions of routine `malloc` without knowing the actual code heap spec Ψ, we add premises to the well-formedness theorem of `malloc` similarly as we did for `free`. The specification in CAP is as follows:

$$\forall Pred. \forall Pred'. \forall s_0. \forall \Psi. \forall \mathtt{f}.\ SIDE \wedge (POST \Longrightarrow \Psi(\mathtt{f})) \wedge (A_{init} \Longrightarrow \Psi(\mathtt{init})) \\ \supset \Psi \vdash \{PRE \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}\, \mathbb{C}(\mathtt{malloc})$$

where $\mathbb{C}$(`malloc`) is the code block labeled `malloc`, universally quantified $Pred$, $Pred'$ and s_0 occurs inside the macros PRE, $POST$ and $SIDE$, `init` is the label of a code block that `malloc` refers to, and A_{init} is the assertion established when `malloc` jumps to `init`. Because `malloc` calls `free` during its execution, we use a different register $\mathtt{r}_1$ to hold the return location for routine `malloc`, due to the lack of certified activation records. The well-formedness of all the other code blocks implementing routine `malloc` are modeled similarly.

Proving these theorems is not much different than proving those of `free`. A tricky part is on the splitting of memory blocks. Similar to coalescing, splitting temporarily breaks certain invariants; thus it is hard to handle in conventional type systems. The annotated `malloc` routine in CAP is shown in Figure 10 of Appendix A as an outline of the proof.

5 Example: `copy` program

With the certified implementation (*i.e.,* code and proof) of `free` and `malloc`, we now implement a certified program `copy`. As shown in Figure 7, this `copy` program takes a pointer to a list as the argument, makes a copy of the list, and disposes the original one.

To make use of the certified routines `free` and `malloc`, we define assertions for the list data structure in Figure 8. ($\mathsf{Pair}\ p\ x\ q$) defines a pair at location p

```
list* copy (list* src) {
   target = prev = nil;
   while (src<>nil) {
      p = malloc(2);                              \\ allocate for a new element
      p->data = src->data, p->link = src->link;        \\ copy an element
      old = src, src = src->link, free(old);           \\ dispose old one
      if (prev == nil) {target = p, prev = p}
      else {prev->link = p, prev=p}                  \\ link in new element
   }
   return target;
}
```

Fig. 7. Pseudo code of copy

which stores values x and q; it carries the fact that it resides inside a "malloced" memory block. (Slist $\alpha\ p\ q$) defines a list with the help of Pair; it represents a list segment from p to q representing the sequence α. The structure of the Slist definition is close to that of MBlkLst and Reynolds' ***singly-linked list*** [19, 18].

The MBlk assertion carried inside Pair is crucial for the memory block to be "freed" when required. It has to be preserved throughout the copy program. Typically when operating on a pair at location p, only locations p and $p+1$ are referred to. Thus as long as the header of the memory block is untouched, preserving MBlk is straightforward.

Certifying the copy program involves the following steps: (1) write the plain code; (2) write the code heap spec; (3) prove the well-formedness of the code with respect to the spec, with the help of the library proofs. Figure 11 of Appendix A shows the copy program with annotations at various program points.

The spec for the code blocks that implement the copy program depends on what property one wants to achieve. In our example, we specify the partial correctness that if copy ever completes its task (by jumping to `halt`), the result list contains the same sequence as the original one.

We get the specs of the library blocks by instantiating the spec templates of the previous section with appropriate assertion *Pred*. The only place where `malloc` is called is in block `nxt0` of copy. Inspecting the assertion at that place and the spec template, we instantiate *Pred* appropriately to get the actual spec. Although `free` is called only once in program copy (in block `nxt1`), it has another call-site in block `more` of `malloc`. Thus for any block of `free`, there are two instantiated specs, one customized for copy (A_1) and the other for `malloc` (A_2). The actual spec that we use is the disjunction of these two ($A_1 \vee A_2$).

The well-formedness of the program can be derived from the well-formedness of all the code blocks. We follow the proof outline in Figure 11 to handle the blocks of copy. For the blocks of routine `malloc`, we directly import their well-formedness theorems described in the previous section. Proving the premises of these theorems (*e.g.*, $A_{init} \Longrightarrow \Psi(\texttt{init})$) is trivial (*e.g.*, A_{init} is exactly $\Psi(\texttt{init})$). For routine `free` whose spec has a disjunction form, we apply Rule 12 to break up the disjunction and apply the theorems twice. Proving the premises of these

$$\mathsf{Pair}\ p\ x\ q \equiv \forall(\mathbb{H},\mathbb{R}).\,\exists lnk.\,\exists siz.\,(\mathbb{H}(p) = x) \wedge (\mathbb{H}(p+1) = q)$$
$$\wedge(\mathsf{MBlk}\ p\ lnk\ siz) \wedge (siz - 2 \geq 2)$$
$$\mathsf{Slist}\ \epsilon\ p\ q \equiv \mathbf{emp} \wedge (p = q)$$
$$\mathsf{Slist}\ (x{\cdot}\alpha)\ p\ q \equiv \exists p'.(\mathsf{Pair}\ p\ x\ p')*(\mathsf{Slist}\ \alpha\ p'\ q)$$

Fig. 8. Pair and Slist.

theorems involves *or-elimination* of the form $A_1 \Longrightarrow A_1 \vee A_2$, which is also trivial. We refer interested readers to our implementation [21] for the exact details.

6 Related Work and Future Work

Dynamic storage allocation Wilson *et al.* [22] categorized allocators based on *strategies* (which attempt to exploit regularities in program behavior), *placement policies* (which decide where to allocate and return blocks in memory), and mechanisms (which involve the algorithms and data structures that implement the policy). We believe that the most tricky part in certifying various allocators is on the low-level mechanisms, rather than the high-level strategies and policies. Most allocators share some subsidiary techniques, such as *splitting* and *coalescing*. Although we only provided a single allocation library implementing a particular policy, the general idea used to certify the techniques of splitting and coalescing can be applied to implement other policies.

Hoare logic Our logic reasonings about memory properties directly follow Reynolds' separation logic [19, 18]. However, being at an assembly level, CAP has some advantages in the context of mechanical proof-checking. CAP provides a fixed number of registers. So the dynamic state is easier to model than using infinite number of variables, and programs are free of variable shadowing. Being at a lower-level implies that the compiler is easier to build, hence it engages a smaller Trusted Computing Base (TCB). Defining assertions as CiC terms of type $\mathsf{State} \rightarrow \mathsf{Prop}$, as opposed to defining assertions syntactically, is also crucial for mechanical proof-checking and thus for PCC. Another difference is that we establish the soundness property using a syntactic approach.

Filliâtre [5, 6] developed a software certification tool *Why* which takes annotated programs as input and outputs proof obligations based on Hoare logic for proof assistants Coq and PVS. It is possible to apply *Why* in the PCC framework, because the proof obligation generator is closely related to the verification condition generator of PCC. However, it is less clear how to apply *Why* to Foundational PCC because the proof obligation generator would have to be trusted. On the other hand, if *Why* is applied to certify memory management, it is very likely to hit problems such as expressing inductively defined assertions. Our treatment of assertions in mechanical proof-checking can be used to help.

Certifying compilation This paper is largely complementary to existing work on certifying compilation [16, 12, 3, 1]. Existing work have only focused on programs whose safety proofs can be automatically generated. On contrast, we support general properties and partial program correctness, but we rely on the programmer to construct the proof. Nevertheless, we believe this is necessary for reasoning about program correctness. Automatic proof construction is infeasible because the problem in general is undecidable. Our language can be used to formally present the reasonings of a programmer. With the help of proof-assistants, proof construction is not difficult, and the result can be mechanically checked.

Future work Exploring the similarity appeared between Hoare-logic systems and type systems, we intend to model types as assertion macros in CAP to ease the certifying task. For instance, a useful macro is the type of a memory block (MBlk). With lemmas (*c.f.,* typing rules) on how this macro interacts with commands, users can propagate it conveniently. If one is only interested in common properties, (*e.g.,* operations are performed only on allocated blocks), it is promising to achieve proof construction with little user directions, or automatically.

In the future, it would be interesting to develop high-level (*e.g.,* C-like or Java-like) surface languages with similar explicit specifications so that programs are written at a higher-level. "Proof-preserving" compilation from those languages to CAL may help retain a small trusted computing base.

7 Conclusion

Existing certifying compilers have only focused on programs whose safety proofs can be automatically generated. In complementary to these work, we explored in this paper how to certify general properties and program correctness in the PCC framework, letting programmers provide proofs with help of proof assistants. In particular, we presented a certified library for dynamic storage allocation — a topic hard to handle using conventional type systems. The logic reasonings on memory management largely follow separation logic. In general, applying Hoare-logic reasonings in the PCC framework yields interesting possibilities.

References

1. A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.
2. A. W. Appel and E. W. Felten. Models for security policies in proof-carrying code. Technical Report CS-TR-636-01, Princeton Univ., Dept. of Computer Science, Mar. 2001.
3. C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, 2000. ACM Press.
4. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
5. J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. Journal of Functional Programming (to appear), 2001.

6. J.-C. Filliâtre. The WHY certification tool, tutorial and reference manual. `http://why.lri.fr/`, July 2002.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.
8. C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, Jan. 1971.
9. W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism.* Academic Press, 1980.
10. B. W. Kernighan and D. M. Ritchie. *The C Programming Language (Second Edition).* Prentice Hall, 1988.
11. D. E. Knuth. *The Art of Computer Programming (Second Edition)*, volume 1. Addison-Wesley, 1973.
12. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, Jan. 1998.
13. G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119, New York, Jan. 1997. ACM Press.
14. G. Necula. *Compiling with Proofs.* PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.
15. G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.
16. G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.*, pages 333–344, New York, 1998.
17. C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.
18. J. C. Reynolds. Lectures on reasoning about shared mutable data structure. IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic, Tandil, Argentina, September 6-13, 2000.
19. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Los Alamitos, California, 2002. IEEE Computer Society.
20. The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, Oct. 2001.
21. The FLINT Project. Coq implementation for certified dynamic storage allocation. `http://flint.cs.yale.edu/flint/publications/cdsa.html`, Oct. 2002.
22. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.
23. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Annotated Programs

free : $\{(\textsf{MBlk}\ \mathit{fptr}\ \_\ \_) * (\textsf{Good}\ \mathit{flist})$
$*\mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
subi $hp, \mathit{fptr}, 2$;
movi $\mathit{prev}, \mathtt{nil}$;
mov p, flist;
$\{(\textsf{MBlk}\ (hp+2)\ \_\ \_) * (\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*\mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (\mathit{prev} = \mathtt{nil})\}$
jd $\mathtt{iter}$;

next : $\{(\textsf{MBlk}\ (hp+2)\ \_\ \_) * (\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*\mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (p \neq \mathtt{nil})$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathtt{nil})\}$
bgt $p, hp, \mathtt{tryh}$;
$\{(\textsf{MBlk}\ (hp+2)\ \_\ \_) * (\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*\mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (p < hp) \wedge (p \neq \mathtt{nil})\}$
mov prev, p;
ld $p, p(\mathtt{link})$;
$\{(\textsf{MBlk}\ (hp+2)\ \_\ \_) * (\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*\mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (\mathit{prev} < hp)\}$
jd $\mathtt{iter}$;

njhi : $\{(\textsf{MBlk}\ (hp+2)\ \_\ s) * (\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*\mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (p > hp + s)$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathtt{nil})\}$
st $hp(\mathtt{link}), p$;
$\{(\textsf{MBlk}\ (hp+2)\ p\ s) * (\textsf{EndL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*(\textsf{MBlkLst}\ n\ p\ \mathtt{nil}) * \mathit{Pred}$
$\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (hp < p)$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathit{nil})\}$
jd $\mathtt{tryl}$;

tryl : $\{(\textsf{EndL}\ \mathit{flist}\ \mathit{prev}\ \_) * \mathit{Pred}$
$*(\textsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil})$
$\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (\mathit{prev} < hp \vee \mathit{prev} = \mathtt{nil})\}$
bgti $\mathit{prev}, \mathtt{nil}, \mathtt{lnkl}$;
$\{(\textsf{EndL}\ \mathit{flist}\ \mathit{prev}\ \_) * \mathit{Pred}$
$*(\textsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil})$
$\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (\mathit{prev} = \mathtt{nil})\}$
mov flist, hp;
$\{(\textsf{Good}\ \mathit{flist}) * \mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
jmp $\mathtt{r}_0$;

njlo : $\{(\textsf{MBlkLst}\ m\ \mathit{flist}\ \mathit{prev})$
$*(\textsf{MBlk}\ (\mathit{prev}+2)\ \_\ \mathit{prevsize})$
$*(\textsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil}) * \mathit{Pred}$
$\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (\mathit{prev} < hp)$
$\wedge(\mathit{prev} + \mathit{prevsize} < hp)\}$
st $\mathit{prev}(\mathtt{link}), hp$;
$\{(\textsf{Good}\ \mathit{flist}) * \mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
jmp $\mathtt{r}_0$;

iter : $\{(\textsf{MBlk}\ (hp+2)\ \_\ \_) * \mathit{Pred}$
$*(\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathtt{nil})\}$
bgti $p, \mathtt{nil}, \mathtt{next}$;
st $hp(\mathtt{link}), p$;
jd $\mathtt{tryl}$;

tryh : $\{(\textsf{MBlk}\ (hp+2)\ \_\ \_) * \mathit{Pred}$
$*(\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
$\wedge(p \neq \mathtt{nil}) \wedge (hp < p)$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathit{nil})\}$
ld $\mathit{cursize}, hp(\mathtt{size})$;
add $\mathit{curend}, hp, \mathit{cursize}$;
bgt $p, \mathit{curend}, \mathtt{njhi}$;
$\{(\textsf{MBlk}\ (hp+2)\ \_\ \mathit{cursize}) * \mathit{Pred}$
$*(\textsf{MidL}\ \mathit{flist}\ \mathit{prev}\ p) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathtt{nil})$
$\wedge(p = hp + \mathit{cursize} \neq \mathtt{nil})\}$
ld $\mathit{psize}, p(\mathtt{size})$;
add $\mathit{newsize}, \mathit{cursize}, \mathit{psize}$;
st $hp(\mathtt{size}), \mathit{newsize}$;
ld $\mathit{plink}, p(\mathtt{link})$;
st $hp(\mathtt{link}), \mathit{plink}$;
$\{(\textsf{MBlk}\ (hp+2)\ q\ \_) * \mathit{Pred}$
$*(\textsf{EndL}\ \mathit{flist}\ \mathit{prev}\ p)$
$*(\textsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
$\wedge(\mathit{prev} < hp \vee \mathit{prev} = \mathit{nil})$
$\wedge(hp < q \vee q = \mathtt{nil})\}$
jd $\mathtt{tryl}$;

lnkl : $\{(\textsf{MBlkLst}\ m\ \mathit{flist}\ \mathit{prev})$
$*(\textsf{MBlk}\ (\mathit{prev}+2)\ \_\ s) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
$*(\textsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil}) * \mathit{Pred}$
$\wedge(\mathit{prev} < hp) \wedge (\mathit{prev} \neq \mathtt{nil})\}$
ld $\mathit{prevsize}, \mathit{prev}(\mathtt{size})$;
add $\mathit{prevend}, \mathit{prev}, \mathit{prevsize}$;
bgt $hp, \mathit{prevend}, \mathtt{njlo}$;
$\{(\textsf{MBlkLst}\ m\ \mathit{flist}\ \mathit{prev})$
$*(\textsf{MBlk}\ (\mathit{prev}+2)\ \_\ \mathit{prevsize})$
$*(\textsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil}) * \mathit{Pred}$
$\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (\mathit{prev} < hp)$
$\wedge(\mathit{prev} + \mathit{prevsize} = hp)\}$
ld $\mathit{cursize}, hp(\mathtt{size})$;
add $\mathit{newsize}, \mathit{prevsize}, \mathit{cursize}$;
ld $\mathit{curlink}, hp(\mathtt{link})$;
st $\mathit{prev}(\mathtt{size}), \mathit{newsize}$;
st $\mathit{prev}(\mathtt{link}), \mathit{curlink}$;
$\{(\textsf{Good}\ \mathit{flist}) * \mathit{Pred} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
jmp $\mathtt{r}_0$;

where $\mathtt{link} \equiv 0$, $\mathtt{size} \equiv 1$ and $\mathtt{nil} \equiv 0$; variables are shorthands for registers.

Fig. 9. Annotated program of free.

$\mathtt{malloc}: \{Pred * (\mathsf{Good}\ flist)$
$\quad \wedge (nsize = s_0 > 0) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{addi}\ bsize, nsize, 2;$
$\quad \mathsf{jd}\ \mathtt{init};$

$\mathtt{init}: \{Pred * (\mathsf{Good}\ flist)$
$\quad \wedge (0 < s_0 + 2 \leq bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{movi}\ prev, \mathtt{nil};$
$\quad \mathsf{mov}\ p, flist;$
$\quad \{Pred * (\mathsf{MidL}\ flist\ prev\ p)$
$\quad \wedge (0 < s_0 + 2 \leq bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{jd}\ \mathtt{miter};$

$\mathtt{miter}: \{Pred * (\mathsf{MidL}\ flist\ prev\ p)$
$\quad \wedge (0 < s_0 + 2 \leq bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{bgti}\ p, \mathtt{nil}, \mathtt{comp};$
$\quad \mathsf{bgt}\ NALLOC, bsize, \mathtt{mod};$
$\quad \{Pred * (\mathsf{Good}\ flist)$
$\quad \wedge (0 < s_0 + 2 \leq bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{mov}\ bbsize, bsize;$
$\quad \mathsf{jd}\ \mathtt{more};$

$\mathtt{comp}: \{Pred * (\mathsf{MidL}\ flist\ prev\ p) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (0 < s_0 + 2 \leq bsize) \wedge (p \neq \mathtt{nil})\}$
$\quad \mathsf{ld}\ psize, p(\mathtt{size});$
$\quad \mathsf{addi}\ ebsize, bsize, 2;$
$\quad \mathsf{bgt}\ psize, ebsize, \mathtt{split};$
$\quad \{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
$\quad * (\mathsf{MBlk}\ (p+2)\ q\ psize) * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil})$
$\quad \wedge (0 < s_0 + 2 \leq bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (p < q \vee q = \mathtt{nil})\}$
$\quad \mathsf{bgt}\ bsize, psize, \mathtt{mnext};$
$\quad \{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
$\quad * (\mathsf{MBlk}\ (p+2)\ q\ psize)$
$\quad * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil})$
$\quad \wedge (s_0 + 2 \leq bsize \leq psize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (p < q \vee q = \mathtt{nil})\}$
$\quad \mathsf{ld}\ plink, p(\mathtt{link});$
$\quad \mathsf{bgti}\ prev, \mathtt{nil}, \mathtt{lprv};$
$\quad \{Pred * (\mathsf{EndL}\ flist\ \mathtt{nil}\ p)$
$\quad * (\mathsf{MBlk}\ (p+2)\ plink\ psize)$
$\quad * (\mathsf{MBlkLst}\ n\ plink\ \mathtt{nil})$
$\quad \wedge (s_0 + 2 \leq psize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{mov}\ flist, plink;$
$\quad \{Pred * (\mathsf{MBlk}\ (p+2)\ plink\ psize)$
$\quad * (\mathsf{MBlkLst}\ n\ flist\ \mathtt{nil})$
$\quad \wedge (s_0 + 2 \leq psize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{jd}\ \mathtt{retptr};$

$\mathtt{retptr}: \{Pred * (\mathsf{Good}\ flist) * (\mathsf{MBlk}\ (p+2)\ \_\ s)$
$\quad \wedge (s_0 + 2 \leq s) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{addi}\ mptr, p, 2;$
$\quad \{Pred' * (\mathsf{Good}\ flist) * (\mathsf{MBlk}\ mptr\ \_\ s)$
$\quad \wedge (s_0 + 2 \leq s) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{jmp}\ \mathtt{r}_1;$

$\mathtt{mod}: \{Pred * (\mathsf{Good}\ flist) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f} \wedge$
$\quad (0 < s_0 + 2 \leq bsize < NALLOC)\}$
$\quad \mathsf{mov}\ bbsize, NALLOC;$
$\quad \mathsf{jd}\ \mathtt{more};$

$\mathtt{split}: \{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
$\quad * (\mathsf{MBlk}\ (p+2)\ q\ psize)$
$\quad * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (0 < s_0 + 2 \leq bsize < psize - 2)$
$\quad \wedge (p < q \vee q = \mathtt{nil})\}$
$\quad \mathsf{sub}\ psize, psize, bsize;$
$\quad \mathsf{st}\ p(\mathtt{size}), psize;$
$\quad \mathsf{add}\ p, p, psize;$
$\quad \mathsf{st}\ p(\mathtt{size}), bsize;$
$\quad \{Pred * (\mathsf{EndL}\ flist\ prev\ p')$
$\quad * (\mathsf{MBlk}\ (p'+2)\ q\ (psize - bsize))$
$\quad * (\mathsf{MBlk}\ (p+2)\ \_\ bsize)$
$\quad * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (0 < s_0 + 2 \leq bsize < psize - 2)$
$\quad \wedge (p' < q \vee q = \mathtt{nil})\}$
$\quad \mathsf{jd}\ \mathtt{retptr};$

$\mathtt{mnext}: \{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
$\quad * (\mathsf{MBlk}\ (p+2)\ q\ s)$
$\quad * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (0 < s_0 + 2 \leq bsize)$
$\quad \wedge (p < q \vee q = \mathtt{nil})\}$
$\quad \mathsf{mov}\ prev, p;$
$\quad \mathsf{ld}\ p, p(\mathtt{link});$
$\quad \{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
$\quad * (\mathsf{MBlkLst}\ n\ p\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (0 < s_0 + 2 \leq bsize)\}$
$\quad \mathsf{jd}\ \mathtt{miter};$

$\mathtt{lprv}: \{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
$\quad * (\mathsf{MBlk}\ (p+2)\ plink\ s) *$
$\quad (\mathsf{MBlkLst}\ n\ plink\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (s_0 + 2 \leq s) \wedge (prev \neq \mathtt{nil})$
$\quad \wedge (p < plink \vee plink = \mathtt{nil})\}$
$\quad \mathsf{st}\ prev(\mathtt{link}), plink;$
$\quad \{Pred * (\mathsf{EndL}\ flist\ prev\ plink)$
$\quad * (\mathsf{MBlk}\ (p+2)\ plink\ s)$
$\quad * (\mathsf{MBlkLst}\ n\ plink\ \mathtt{nil})$
$\quad \wedge (s_0 + 2 \leq s) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
$\quad \mathsf{jd}\ \mathtt{retptr};$

$\mathtt{more}: \{Pred * (\mathsf{Good}\ flist) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
$\quad \wedge (0 < s_0 + 2 \leq bsize \leq bbsize)\}$
$\quad \mathsf{alloc}\ newp[bbsize];$
$\quad \mathsf{st}\ newp(\mathtt{size}), bbsize;$
$\quad \mathsf{addi}\ fptr, newp, 2;$
$\quad \mathsf{movi}\ \mathtt{r}_0, \mathtt{init};$
$\quad \{(\mathsf{Good}\ flist) * (\mathsf{MBlk}\ fptr\ \_\ bbsize)$
$\quad * Pred \wedge (0 < s_0 + 2 \leq bsize)$
$\quad \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{init}\}$
$\quad \mathsf{jd}\ \mathtt{free};$

where $\mathtt{link} \equiv 0$, $\mathtt{size} \equiv 1$ and $\mathtt{nil} \equiv 0$; variables are shorthands for registers.

Fig. 10. Annotated program of `malloc`.

```
copy : {∃α. (Slist α src nil) ∗ (Good flist)
        ∧(α = α_0)}
       movi tgt, nil;
       movi cprev, nil;
       jd test;

test : {(((cprev = nil)
          ⊃ ∃α. (Slist α src nil) ∧ (α = α_0)
            ∧(tgt = nil))
        ∧((cprev ≠ nil)
          ⊃ ∃α. ∃β. ∃b. (β·b·α = α_0)
            ∧(Slist α src nil) ∗ (Slist β tgt cprev)
            ∗(Pair cprev b src)))
        ∗(Good flist)}
       bgti src, nil, nxt0;
       jd halt;

nxt0 : {(((cprev = nil)
          ⊃ ∃a. ∃α. ∃i. (Pair src a i) ∗ (Slist α i nil)
            ∧(a·α = α_0) ∧ (tgt = nil))
        ∧((cprev ≠ nil)
          ⊃ ∃a. ∃α. ∃β. ∃b. ∃i. (β·b·a·α = α_0)
            ∧(Pair src a i) ∗ (Slist α i nil)
            ∗(Slist β tgt cprev)
            ∗(Pair cprev b src)))
        ∗(Good flist)}
       movi nsize, 2;
       movi r_1, nxt1;
       {(((cprev = nil)
          ⊃ ∃a. ∃α. ∃i. (Pair src a i) ∗ (Slist α i nil)
            ∧(a·α = α_0) ∧ (tgt = nil))
        ∧((cprev ≠ nil)
          ⊃ ∃a. ∃α. ∃β. ∃b. ∃i. (β·b·a·α = α_0)
            ∧(Pair src a i) ∗ (Slist α i nil)
            ∗(Slist β tgt cprev)
            ∗(Pair cprev b src)))
        ∗(Good flist) ∧ (nsize = 2) ∧ (r_1 = nxt1)}
       jd malloc;

lnkp : {(((cprev ≠ nil)
          ∧ ∃a. ∃α. ∃β. ∃b. (Pair mptr a src)
            ∗(Slist α src nil) ∗ (Slist β tgt cprev)
            ∗(Pair cprev b fptr) ∧ (β·b·a·α = α_0)))
        ∗(Good flist)}
       st cprev(1), mptr;
       mov cprev, mptr;
       {(((cprev ≠ nil)
          ∧ ∃a. ∃α. ∃β. (Pair cprev a src)
            ∗(Slist α src nil)
            ∗(Slist β tgt cprev) ∧ (β·a·α = α_0)))
        ∗(Good flist)}
       jd test;

halt : {∃β. (Slist β tgt nil) ∗ (Good flist)
        ∧(β = α_0)}
       jd halt;

nxt1 : {(((cprev = nil)
          ⊃ ∃a. ∃α. ∃i. (Pair src a i)
            ∗(Slist α i nil) ∧ (a·α = α_0)
            ∧(tgt = nil))
        ∧((cprev ≠ nil)
          ⊃ ∃(a, α, β, b, i). (β·b·a·α = α_0)
            ∧(Pair src a i) ∗ (Slist α i nil)
            ∗(Slist β tgt cprev)
            ∗(Pair cprev b src)))
        ∗(Good flist) ∗ (MBlk mptr _ siz)
        ∧(siz ≥ 4)}
       ld temp, src(0);
       st mptr(0), temp;
       mov fptr, src;
       ld src, src(1);
       st mptr(1), src;
       movi r_0, nxt2
       {(((cprev = nil)
          ⊃ ∃a. ∃α. (Pair mptr a src)
            ∗(Slist α src nil) ∧ (a·α = α_0)
            ∧(tgt = nil))
        ∧((cprev ≠ nil)
          ⊃ ∃a. ∃α. ∃β. ∃b. (β·b·a·α = α_0)
            ∧(Pair mptr a src)
            ∗(Slist α src nil)
            ∗(Slist β tgt cprev)
            ∗(Pair cprev b fptr)))
        ∗(MBlk fptr _ _) ∗ (Good flist)
        ∧(r_0 = nxt2)}
       jd free;

nxt2 : {(((cprev = nil)
          ⊃ ∃a. ∃α. (Pair mptr a src)
            ∗(Slist α src nil) ∧ (a·α = α_0)
            ∧(tgt = nil))
        ∧((cprev ≠ nil)
          ⊃ ∃a. ∃α. ∃β. ∃b. (β·b·a·α = α_0)
            ∧(Pair mptr a src)
            ∗(Slist α src nil)
            ∗(Slist β tgt cprev)
            ∗(Pair cprev b fptr)))
        ∗(Good flist)}
       bgti cprev, nil, lnkp;
       {∃a. ∃α. (Pair mptr a src)
        ∗(Slist α src nil) ∧ (a·α = α_0)
        ∧(tgt = nil)
        ∗(Good flist) ∧ (cprev = nil)}
       mov tgt, mptr;
       mov cprev, tgt;
       {∃a. ∃α. (Pair cprev a src)
        ∗(Slist α src nil) ∧ (a·α = α_0)
        ∧(cprev = tgt)
        ∗(Good flist) ∧ (cprev ≠ nil)}
       jd test;
```

where nil ≡ 0; variables are shorthands for registers.

Fig. 11. Annotated program of copy: copies a null-terminated list from *src* to *tgt*.